CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Towards Certifiable Adaptive Reservations for Hypervisor-based Virtualization

Stefan Groesbrink

Luis Almeida

Mario de Sousa

Stefan M. Petters

CISTER-TR-140304

Version:

Date: 3/10/2014

Towards Certifiable Adaptive Reservations for Hypervisor-based Virtualization

Stefan Groesbrink, Luis Almeida, Mario de Sousa, Stefan M. Petters

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: , lda@det.ua.pt, , smp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Hypervisor-based virtualization provides a natural way to integrate formerly distinct systems into a single mixedcriticality multicore system by consolidating in separated virtual machines. We propose an adaptive computation bandwidth management for such architectures, which is compatible with a potential certification based on the guarantee of specified bandwidth minimums and the isolation of overruns of virtual machines. This management uses periodic servers and an elastic task model to combine analyzability at design time with adaptability at runtime. Mode changes or early termination of VMs trigger a resource redistribution that reassigns spare capacity. In this paper we focus on the integration of an adaptive reservation policy into a virtualization software stack and the co-design of hypervisor and paravirtualized guest operating system. In a concrete implementation on a PowerPC 405, the bandwidthdistribution policy incurred in a memory footprint below 2.7KB and a worst-case execution time for the redistribution function below 4 microseconds for realistic low numbers of VMs. Simulations over synthetically generated sets of VMs with random mode changes showed a gain of 13% of computation bandwidth when compared to an approach with fixed partitions and provided a relative error of allocated bandwidth to desired bandwidth 4 times lower.

Towards Certifiable Adaptive Reservations for Hypervisor-based Virtualization

Stefan Groesbrink
Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany
s.groesbrink@upb.de

Luis Almeida
IT - Faculty of Engineering
University of Porto
Porto, Portugal
lda@fe.up.pt

Mario de Sousa
INESC TEC
(formerly INESC Porto)
and Faculty of Engineering
University of Porto
Porto, Portugal
msousa@fe.up.pt

Stefan M. Petters
CISTER/INESC-TEC,
ISEP, IPP
Porto, Portugal
smp@isep.ipp.pt

***Abstract*—Hypervisor-based virtualization provides a natural way to integrate formerly distinct systems into a single mixed-criticality multicore system by consolidating in separated virtual machines. We propose an adaptive computation bandwidth management for such architectures, which is compatible with a potential certification based on the guarantee of specified bandwidth minimums and the isolation of overruns of virtual machines. This management uses periodic servers and an elastic task model to combine analyzability at design time with adaptability at runtime. Mode changes or early termination of VMs trigger a resource redistribution that reassigns spare capacity. In this paper we focus on the integration of an adaptive reservation policy into a virtualization software stack and the co-design of hypervisor and paravirtualized guest operating system. In a concrete implementation on a PowerPC 405, the bandwidth distribution policy incurred in a memory footprint below 2.7KB and a worst-case execution time for the redistribution function below 4 microseconds for realistic low numbers of VMs. Simulations over synthetically generated sets of VMs with random mode changes showed a gain of 13% of computation bandwidth when compared to an approach with fixed partitions and provided a relative error of allocated bandwidth to desired bandwidth 4 times lower.**

I. Introduction

Hypervisor-based virtualization is a promising software architecture to meet the high functionality and reliability requirements of complex embedded systems. A hypervisor separates operating system (OS) and hardware in order to share the hardware among multiple virtual machines (VM), each running an isolated instance of an OS that caters for the execution of a subset of applications [1] (Fig. 1). The resulting consolidation of multiple systems with maintained separation and resource partitioning is well-suited to combine independently developed software into a system of systems.

The rise of *multicore* embedded processors [2] is a major enabler for virtualization, whose architectural abstraction eases the migration from single-core to multicore platforms. The replacement of multiple hardware units by a single multicore system has the potential to provide the required computational capacity with reduced size, weight, and power consumption.

Hypervisors support *mixed-criticality* systems by consolidating systems of different criticality levels in separated VMs, each running an adequate OS, for example, an efficient and highly predictive real-time executive for safety-critical control tasks and a feature-rich general purpose OS for the communication or human-machine interface. Moreover, the addition of subsystems of low criticality to critical subsystems can increase significantly the resource utilization, as critical applications suffer from a more pessimistic worst-case resource demand determination [3].

Key requirements for the consolidation of real-time systems and particularly for certification are temporal isolation and a guaranteed minimum bandwidth allocation to all VMs that enables them to meet their timing constraints. Computing these minimum bandwidths goes beyond the scope of this paper and implies addressing topics such as inter-core memory and bus interference that are orthogonal to our work. In this paper we assume knowledge of the bandwidth requirements, even if pessimistic, and we focus on the following aspects of temporal isolation, as explicitly demanded by the standard for functional safety of road vehicles ISO 26262 [4]:

- The hypervisor scheduling guarantees that all guest systems receive sufficient computation time in order to meet their real-time constraints.
- Overruns within a VM provoke under no circumstances that other VMs violate their real-time constraints.

Virtualization solutions including dynamic resource management are state of the art for computing servers. However, these solutions are not applicable to embedded systems, as compliance with real-time requirements cannot be guaranteed. In order to ensure real-time behavior, existing virtualization solutions for embedded systems apply a static computation bandwidth allocation [5]. Typical solutions are a one-to-one mapping of VMs to processors or a cyclic schedule with fixed execution time slices if VMs share a processor. Static resource allocation naturally leads to fragmentation of available resources since reserved but unused capacity cannot be reclaimed to improve the performance of other VMs. In addition, static approaches are inappropriate for the varying resource requirements of adaptive and self-optimizing applications.

In this work, we propose an adaptive bandwidth management based on dynamic budgets of periodic servers combined with an elastic task model. A preliminary proposal with just the conceptual idea was presented in [6]. Next to the

appropriate consideration of multiple criticality levels, one goal is obtaining a desired distribution of spare capacity among the VMs. This is particularly valuable for applications with strongly variable execution time and which can take advantage of higher capacity to produce improved results. The focus is on the integration of an adaptive reservation policy into a software stack and the co-design of hypervisor and paravirtualized guest OS. The evaluation investigates the overhead regarding execution time, memory footprint, and paravirtualization effort. The benefits compared to a fixed-bandwidth allocation are shown with simulations of synthetic workloads.

A. Contribution & Organization

The contribution of this work is the discussion of fundamental implications of hypervisor-based virtualization for the design of an adaptive computation bandwidth management for multi-mode mixed-criticality systems. This management follows an elastic approach and we propose a new bandwidth distribution algorithm that is non-iterative, being more predictable and faster than the existing iterative solutions in the literature. Moreover, we address the co-design of hypervisor and operating system, the argumentation that paravirtualization is required, and a discussion of implementation issues for both hypervisor and operating system. The overheads are determined with an implementation on real low-performance embedded hardware and highlight the feasibility of the approach. An experimental evaluation underlines the benefit compared to a fixed-bandwidth allocation.

In the remainder of this paper we discuss related work (Sec. II), present a motivating application example (Sec. III), make a case for paravirtualization (Sec. IV), introduce the system model (Sec. V), propose a dynamic bandwidth distribution policy (Sec. VI), and finally evaluate the proposal (Sec. VII).

II. Related Work

Feedback-control algorithms for adaptive reservations [7] [8] measure the performance of the served tasks and adjust the budgets according to a certain control law. Khalilzad et al. introduced a hierarchical single-core scheduling framework that modifies the budgets of periodic servers after a deadline miss (overload situation) based on the amount of idle time [9]. Block et al. presented an adaptive multiprocessor scheduling framework, which adjusts processor shares in order to maximize the quality of service (QoS) of soft real-time tasks [10].

Buttazzo et al. introduced an adaptive multicore resource management for smartphones [11], which selects a service level for each application by solving an integer linear programming problem. Maggio et al. recently proposed a game-theoretic approach for the resource management among competing QoS-aware applications, which decouples service level assignment and resource allocation [12]. Applications do not have to inform the resource manager about the available service levels, but about the start and stop time of each job.

Zabos et al. presented the integration of a spare reclamation algorithm into a middleware layer [13] that is placed on top of a real-time OS, and not underneath as a hypervisor. Dynamic reclaiming algorithms such as GRUB [14] or BASH [15] take advantage of spare bandwidth when tasks do not need their WCET and distribute it in a greedy or weighted manner, as proposed in this work.

Nogueira and Pinho proposed a dynamic scheduler for the coexistence of isolated and non-isolated servers [16]. An isolated server obtains a guaranteed budget, whereas budget can be stolen from a non-isolated server. In order to avoid the increased computational complexity of a fair distribution, the entire slack is assigned to the currently executing server. Bernat and Burns proposed as well a budget stealing server-based scheduling [17]. Each server handles a single soft task and can in overload situations steal budget from the other servers. Temporal isolation is lost and a server of low priority might receive less bandwidth than requested.

IRIS is a resource reservation algorithm that handles overload situations by spare bandwidth allocation among hard, soft, and non-real-time tasks [18]. As it is the case for our approach, minimum budgets are guaranteed and the remaining bandwidth is distributed in a fair manner among the servers. Conversely to the proposal in this paper, the scheduling is based on an extension of CBS and EDF. In the context of the ACTORS EU project, an adaptive reservation-based multicore CPU management for soft real-time systems was developed, as well based on CBS and EDF [19]. Similar to our work, a partitioned hierarchical scheduler is proposed, however one for the OS (implemented in Linux), handling groups of threads.

Su and Zhu proposed an elastic mixed-criticality task model and an EDF-based uniprocessor scheduling [20]. Slack is passed at runtime to low-criticality tasks, based on variable periods. Anderson et al. presented the first work on server-based mixed-criticality multicore scheduling [21]. On each core, budget is specified for each criticality level and consumed in parallel corresponding to the respective level. Mollison et al. introduced the notion of higher-criticality tasks as slack generators for lower-criticality tasks [22]. Herman et al. presented the first implementation of an OS's mixed-criticality multicore scheduler and discussed design tradeoffs [23]. Their framework reclaims capacity lost to WCET pessimism.

All so far cited approaches are concerned with the OS's resource management among applications, and do not target hypervisor-based virtualization. In contrast, Bruns et al. evaluated virtualization to consolidate subsystems of mobile devices on a single processor [24]. The software stack consists of an L4/Fiasco microkernel and a paravirtualized Linux. Crespo et al. designed XtratuM, a hypervisor for the avionics domain with a fixed cyclic scheduling [25]. A redesign for multicore processors was recently published [26]. Yang et al. proposed a compositional scheduling framework for virtualization without slack distribution based on the L4/Fiasco microkernel [27]. Cucinotta et al. examined hard reservations and an EDF-based soft real-time scheduling policy to provide temporal isolation among I/O-intensive and CPU-intensive VMs [28]. Their implementation is based on the Linux kernel module Kernel-based Virtual Machine (KVM).

Closest to our work, Lee et al. presented a compositional scheduling framework for the Xen hypervisor [29]. Resource models are realized as periodic servers and enhancements to the server design in order to increase the resource utilization are introduced. Their work-conserving periodic server lets one lower-priority non-idle server benefit when a high-priority

server idles. Their capacity reclaiming periodic server allows idle time of a server to be used by any other server. Their work was very influential for us and we use their results on periodic resource model design for quantum-based platforms. In contrast to this work, their slack distribution does not consider fairness and they target application domains with powerful hardware and timing requirements in the range of milliseconds (scheduling quantum of 1ms), whereas our work targets low-performance and memory-constrained embedded hardware with timing requirements in the sub-millisecond range. In addition, our approach includes the distribution of structural and dynamic slack.

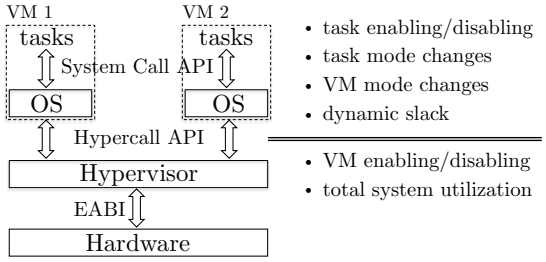


Fig. 1: Availability of Scheduling Information on the Different Levels of the Software Stack

III. A Motivating Application Example

There is a trend reversal for automotive architectures: functions are consolidated on multicore processors instead of following the "one function per ECU" design paradigm [30]. This leads in many cases to a coexistence of systems with different criticality levels and resource requirement characteristics. For example, safety-critical driver assistance systems are integrated with QoS-aware infotainment and computer vision systems within the center stack computer. There is an increased potential to temporarily deactivate functionality that is not in constant use while the car is operational [31]. Different operational modes could be defined by parking/driving/start-stop, charging, or the use of navigation and different multimedia functionality. Instead of turning off control units temporarily, one can consolidate functionality in VMs and deactivate a VM.

Examples for computer vision applications are the detection of objects for a collision warning system or the detection of lanes for a lane-departure warning system. The QoS is directly related to the number of processed frames per second. The execution time of the vision algorithms varies depending on the situation and illumination/weather conditions. Such applications typically benefit from additional resource allocations and both mode changes and task enabling/disabling can be applied subject to the driving situation. The rear view camera system must only be enabled when the car is reversing or the control of an adjustable driver seat has a computation demand only if the car is stopped.

The proposed bandwidth management approach meets the requirements of such systems. Its key characteristics are a flexible workload model, a guarantee of minimum bandwidths, and a dynamic redistribution of bandwidth in the case of either a short-term underutilization caused by varying execution time or of a mode change within a VM to a mode with different computation time demand. The resources of critical systems are in general over-provisioned in a very pessimistic manner to realize a great level of assurance against failures. As a result, the reserved bandwidth is in practice often underutilized and the critical VMs serve as generators of slack, of which the dynamic distribution makes effective use. The guaranteed minimum bandwidths, which are not touched by the dynamic bandwidth management, are the basis for both the schedulability analysis and a potential certification. This aspect is covered in detail in Section VI-D.

The bandwidth allocation is realized cooperatively by guest OS and hypervisor. The guest OS informs about changes and triggers the hypervisor to evaluate whether a bandwidth adaptation is reasonable on the basis of the new situation. The hypervisor informs the guest OS about the result. The implementation requires therefore the awareness of a guest OS that it is executing on top of a hypervisor, a technique known as paravirtualization, discussed in detail in the next section.

IV. The Case for Paravirtualization

The capability to host virtualization-unaware operating systems classifies hypervisors. If unmodified guest OSs can be hosted, it is *full* (or *transparent*) *virtualization*. Conversely, if the guest OS requires porting to the hypervisor's application programming interface (API), it is called *paravirtualization* [32]. In this case, the guest OS is aware of being executed within a VM and uses *hypercalls* to request hypervisor services. Paravirtualization is the prevailing approach in the embedded domain [5]. The need to modify the guest OS is outweighed by the advantages in terms of efficiency (reduction of the overhead [33]) and in terms of run-time flexibility of an explicit communication and the hereby facilitated cooperation of hypervisor and guest OS. The major drawback is the need to port an OS, which involves modifications of critical kernel parts. If legal or technical issues preclude this for an OS, it is not possible to host it. For both kinds of virtualization, the applications executed by the OS do not have to be modified.

The implementation of the proposed adaptive bandwidth allocation requires modifications of the OS, and thereby paravirtualization, since explicit communication between hypervisor and guest OS is mandatory. The OS has to provide the hypervisor a certain level of insight in order to support the hypervisor's bandwidth assignment. The hypervisor in turn informs the guest OSs about the assigned bandwidth share, since they need this information in order to distribute the bandwidth among their tasks. Figure 1 illustrates the different levels of the system stack, the interfaces between them, and the availability of scheduling related information above and below the border between hypervisor and guest OS. Paravirtualization is mandatory to inform the hypervisor about adaptation triggering events such as a task enabling/disabling or mode change. In addition, instead of running the idle task, a paravirtualized guest OS can yield to allow the hypervisor to execute another ready VM. Comparably, Kiszka paravirtualized Linux in order to give the hypervisor (Linux with KVM) a hint about the internal states of its guests [34].

V. System Model

A. Workload Model

Our workload is composed of a set V of n virtual machines as defined in Expression 1, where each virtual machine V_i is defined by a set of tasks $\Omega(V_i)$, a task scheduler $\sigma(V_i)$, a criticality level $\chi(V_i)$, and a QoS parameter $0 < qos(V_i) \leq 1$.

$$V \equiv \{V_i : V_i(\Omega(V_i), \sigma(V_i), \chi(V_i), qos(V_i)), i = 1..n\} \quad (1)$$

From the point of view of the hypervisor, we consider the VMs to be schedulable entities that we represent combining the traditional periodic real-time task model of Liu and Layland [35] and the elastic resource distribution framework of Marau et al. [36]. In particular, each V_i will be taken as a *task* that requires a minimum computation bandwidth $U_{min}(V_i)$ to carry out the timely execution of its internal task set $\Omega(V_i)$. However, it may benefit from additional bandwidth, if available, up to $U_{lax}(V_i) \geq 0$. The distribution of the currently available extra bandwidth, i.e., *processing slack*, among the active VMs is carried out firstly according to the criticality level $\chi(V_i)$ in a greedy manner (highest criticality first). The criticality levels denote different severity of failure and might be associated with a concrete certification level. Secondly, among VMs of the same criticality level, the distribution is weighted with $qos(V_i)$ following the elastic framework in [36]. In this work we adapt this framework to system virtualization. A similar model that combines mandatory requirements with QoS aspects was used by Emberson and Bate [37], however, in the context of fault-tolerance-focused task allocation.

The required minimum utilization $U_{min}(V_i)$ and the maximum extra bandwidth $U_{lax}(V_i)$ are dependent on the task set $\Omega(V_i)$ and on the task scheduling policy $\sigma(V_i)$ and derived in the context of the periodic resource model design (Section V-B). The flexibility of this workload model allows accommodating safety-critical VMs that have a constant resource requirement ($U_{min}(V_i) > 0$, $U_{lax}(V_i) = 0$), VMs that demand a certain guaranteed bandwidth and can benefit from additional computation capacity ($U_{min}(V_i) > 0$, $U_{lax}(V_i) > 0$), and background VMs that have no guaranteed bandwidth ($U_{min}(V_i) = 0$, $U_{lax}(V_i) > 0$). We assume independent VMs with neither shared resources except from the processor, nor data dependencies, nor inter-VM communication.

B. Resource Model

The target platforms are homogeneous multicore systems, consisting of identical cores of equal computing power. This implies that each task has the same execution speed and utilization on each processor core. A virtual processor is a representation of a share of the physical processor to a VM and multiple virtual processors can be mapped onto a single physical processor and receive a fraction of the available bandwidth. If so, a continuous progress of multiple VMs cannot be achieved in practice, but approximated.

The *periodic resource model* $\Gamma(\Pi, \Theta)$ of Shin and Lee [38] provides a formal abstraction of the resource supply by a virtual processor. It models a resource that is available at its full capacity at times and not available at all otherwise. A guaranteed execution budget of Θ time units is allocated every period Π ($0 \leq \Theta \leq \Pi$). The minimum computation time allocation that a virtual processor provides in a time interval of length t is specified in terms of the supply bound function $sbf(t)$. This function can be designed to enforce schedulability of the task set within a VM ($\Omega(V_i)$) by forcing it to be at least as high as the demand bound function $dbf(t)$ of the task set for all t [38]. Moreover, a real implementation must deal with scheduling quantum and temporal granularity. Lee et al. provided an extension of the compositional scheduling frameworks for quantum-based platforms [29]. The server design problem is, however, beyond the scope of this paper.

Each virtual processor $\Gamma_i(\Pi_i, \Theta_i)$ is implemented as a periodic server, characterized by a period and an execution time budget. At the beginning of each period, the budget is replenished by the hypervisor. If scheduled and therefore active, a server's budget is used to execute the computation time demand of the associated VM. The budget is consumed at the rate of one per time unit and once exhausted, the server is not ready for execution until the next period. There is no cumulation of budget from period to period. Such a server enforces a guaranteed, but bounded computation time a VM receives in a specified time span, even in the presence of overloads internal to the VM.

The parameters of the resulting periodic resource model $\Gamma_i(\Pi_i, \Theta_i)$ determine the minimal bandwidth of a VM:

$$U_{min}(V_i) = \frac{\Theta_i}{\Pi_i} \quad (2)$$

C. Partitioned Hierarchical Scheduling

There are two main multiprocessor scheduling approaches, partitioned and global scheduling [39]. Under *partitioned scheduling*, the tasks are statically allocated to a processor, whereas *global scheduling* uses a global ready queue to dynamically map the jobs to any of the processors. Hypervisor-based virtualization consolidates entire software stacks including an OS, resulting in scheduling decisions on two levels (*hierarchical scheduling* [40], [41]). The hypervisor schedules the VMs and the hosted OSs schedule their tasks according to their own local scheduling policies. This is irreconcilable with a scheduling based on a global task ready queue.

In the context of this work, VMs are statically assigned to processors. Although a dynamic mapping is conceptually and technically possible, a static solution is the option taken by the AUTOSAR consortium [30] and eases certification significantly, due to the lower run-time complexity, the higher predictability, and the wider experience of system designer and certification authority with uniprocessor scheduling. Next to assuring schedulability of the VM set assigned to a processor, the partitioning of the VMs focuses on two goals. Minimizing the overall required computation bandwidth is the first goal, since it determines the number of processors required to host the set of VMs. In addition, a distribution of high-criticality VMs among the processors is targeted. If VMs of differing criticality share a processor, there are in general more possibilities to apply an adaptive bandwidth management, since the low-criticality VMs can benefit from unused reservations of the high-criticality VMs.

After the partitioning, verified scheduling techniques from the uniprocessor domain can be reused. On each core, the hypervisor schedules the assigned servers by static priorities

according to the Rate Monotonic (RM) policy: the higher the request rate of a server (the smaller Π), the higher its priority [35]. Starvation of lower-priority servers is nevertheless precluded, since the budget limits the execution time of the higher-priority servers. For the same reason, a criticality-disregarding priority assignment may be made.

The schedulability of the VMs is guaranteed if the resource capacity U_R is equal to or greater than the least upper bound of the processor utilization U_{lub} of the used scheduling policy. For n servers with arbitrary periods scheduled according to the RM policy, $U_{lub} = n(2^{1/n} - 1)$ [35]. Therefore, for the sake of VMs schedulability, we consider that:

$$\sum_{i=1}^{n} U_{min}(V_i) \leq U_{lub} \leq U_R \tag{3}$$

Using the RM server scheduler has the significant advantage of a low runtime overhead, but the major drawback of a low utilization bound. This, however, can be tackled using harmonic periods, i.e. the period of each server is an exact multiple of the periods of every other server with a shorter period. In this case, RM can utilize the processor up to 100% ($U_{lub} = U_R$). This choice is a realistic one and, in fact, the server design approaches [38] [42] allow the server period to be chosen within a range of possible values with minimal impact on the schedulability of the internal task set, in particular without violating the largest possible VM service delay.

The overhead of VM context switches is not negligible and therefore added to the execution time requirement of the VM. With the applied fixed-priority assignment, each VM preempts at most one VM, since VMs do not perform self-blocking and resume later. The overhead is included by adding the execution time for two context switches, one at the start and one at the completion of a VM's execution. If a VM is preempted, the context switch overhead is accounted for through the execution time of the preempting VM. This approach is pessimistic but safe, since it assumes that every instance of an executing VM causes preemption, which is not necessarily the case.

D. Summary

The system model includes the following aspects:

- Partitioned RM scheduling of VMs based on periodic servers with fixed period, but variable budget
- VMs characterized by minimum and maximum computation bandwidth (flexible workload model), criticality level, QoS parameter
- Periodic resource model as an abstraction of the supply by a shared processor (Shin and Lee [38])

VI. Dynamic Bandwidth Distribution

The bandwidth adaptation is realized by a dynamic setting of the servers' replenishment budgets. Their harmonic periods are always kept fixed, though, which simplifies analysis and enhances schedulability. Moreover, since VMs are statically assigned to the processors, bandwidth redistribution has to be handled separately for each processor.

Two situations lead (potentially) to a bandwidth adaptation:

- *Distribution of Structural Slack*:
  Many events have a significant and lasting impact on the resource utilization and trigger therefore a redistribution of the spare bandwidth U_{spare}. Those events are an enabling or disabling of a task or VM, or a mode change to a mode with differing resource demand (we assume that modes differ regarding U_{lax}).
- *Distribution of Dynamic Slack*:
  Task execution times are bounded by the static WCET but vary at runtime. When the actual execution time of a task is considerably smaller than the WCET, the difference is termed dynamic slack. Especially in the case of critical tasks, the pessimistic WCET is often not reached, but has to be reserved. Instead of wasting bandwidth when a VM does not demand the allocated share, the guest OS yields and the hypervisor reassigns the reserved but no longer required bandwidth. However, this bandwidth needs to be recovered by the respective VM upon its next periodic reactivation. Thus, dynamic slack distribution is carried out whenever VMs yield and when they are periodically re-triggered.

Nevertheless, bandwidth adaptations incur in a certain overhead and thus, the guest OSs trigger the hypervisor upon every potential adaptation to evaluate whether it is reasonable. The adaptation is taken only if the slack compensates the overhead cost. This cost can be determined for each specific hardware platform and used at runtime as a threshold.

Moreover, the two bandwidth reassignments differ regarding timescale. Structural changes have a long term impact and are reflected in the system by updating the VM set V accordingly, e.g., adding/removing VMs or updating their U_{lax} parameter. The dynamic slack distribution is a short term measure, potentially occurring at every server termination and reactivation. Since it uses the current set V, dynamic and structural slack are distributed jointly.

A. Distributing Bandwidth

At each bandwidth distribution point the hypervisor subdivides the set V into two sets: V' with all currently ready/running VMs; and $\overline{V}$ with the VMs that already terminated their current instance. Moreover, the hypervisor computes how much bandwidth U_{act} was actually used by the VMs in $\overline{V}$. The current slack, which we call *spare bandwidth* U_{spare}, is finally defined as in Eq. 4. Note that $\sum_{\forall V_j \in \overline{V}} U_{act}(V_j)$ accounts for the bandwidth already spent at this point and $\sum_{\forall V_i \in V'} U_{min}(V_i)$ accounts for the minimum bandwidth we want to guarantee.

$$U_{spare} = max(0, U_{lub} - (\sum_{\forall V_j \in \overline{V}} U_{act}(V_j) + \sum_{\forall V_i \in V'} U_{min}(V_i))) \tag{4}$$

The actual bandwidth distribution among the VMs in set V' is carried out in two steps. First, the minimum requirement U_{min} is allocated to each VM. Second, the spare bandwidth

U_{spare} is distributed according to an elastic policy (Sec. VI-B) with the objective to satisfy U_{lax} of the ready/running VMs.

The use of U_{lub} when computing U_{spare} allows us enforcing continued schedulability of the VMs in V'. Nevertheless, there are a few aspects to account for. Consider V_a yields with relatively small $U_{act}(V_a)$, this will lead to a large slack. It is possible that V_b in V' uses the entire slack and terminates, moving to $\overline{V}$ with large $U_{act}(V_b)$, close to its maximum $U_{min}(V_b) + U_{lax}(V_b)$. At the same time, it is possible that V_a that created this slack originally becomes ready again, moving to V' and requesting at least $U_{min}(V_a)$ bandwidth.

This is a pessimistic situation, since V_b while executing the large $U_{act}(V_b)$, did so on behalf of V_a that yielded soon. Consequently, Eq. 4 may generate values for U_{spare} that are smaller than the real ones, which are always positive. On the other hand, the use of actually measured values of U_{act} precludes generating U_{spare} values that are larger than the real ones. Thus, Eq. 4 is safe.

Another aspect to consider is when to apply the capacity changes that result from the bandwidth management. When reducing capacities, the new capacity can replace the current one immediately. If the VM has already executed in the current instance for more than the new assigned capacity, the VM is terminated and moved to $\overline{V}$. When increasing capacities, the new capacity can be applied immediately if the VM is still in V', i.e., ready/running. If it is already idle, i.e., in $\overline{V}$, the new capacity is applied in the next instance, only.

Finally, note that when a VM executes its capacity to completion, i.e., not yielding, the bandwidth distribution that would result from applying Eq. 4 is exactly the same as before and thus there is no need to invoke a bandwidth redistribution.

B. Distribution Policy

The distribution policy considers two factors, namely criticality level and weight, in this order. Thus, the criticality level χ is the dominant factor and the bandwidth is assigned in a greedy manner in order of decreasing criticality. The highest criticality level obtains as much bandwidth as possible, limited by either the distributable amount U_{spare} or the maximum bandwidth requirement of its VMs (typically, the higher the criticality of a VM, the more likely a large U_{min} and a low U_{spare}, since critical systems are rarely QoS-driven). If there is spare bandwidth left, the next lower criticality level is served and so on. The weights influence the bandwidth assignment among VMs of the same criticality level, since a greedy strategy lacks fairness. The determination of the weight of a VM $w(V_i)$ is based on the normalization of its QoS parameter $qos(V_i)$ considering the current set of VMs V' among which U_{spare} will be distributed:

$$w(V_i) = \frac{qos(V_i)}{\sum_{V_j \in V'} qos(V_j)} \tag{5}$$

Assuming the bandwidth U_{spare} is to be distributed among VMs of similar criticality level (VM set V'), the individual shares $U_{add}(V_i)$ are set to:

$$U_{add}(V_i) = w(V_i) \cdot U_{spare} \tag{6}$$

This results in a total bandwidth assignment of:

$$U(V_i) = U_{min}(V_i) + U_{add}(V_i) \tag{7}$$

This value determines the new server bandwidth and the replenished budget follows as $\Theta_i = U(V_i) \cdot \Pi_i$.

However, using the above distribution may result in a value of U_{add} greater than U_{lax} for some VMs. In this case, their U_{add} is truncated to U_{lax}, and the remaining bandwidth is distributed among the other VMs in the same proportion of their weights. The additional bandwidth distribution may result in more VMs reaching their U_{lax} limit, causing more truncations. This process may continue until there is no remaining bandwidth, resulting in the iterative algorithms proposed in [43] and [36] for the elastic bandwidth management.

C. The Algorithm and its Computational Complexity

The typical iterative approaches followed by elastic management algorithms introduce significant overhead and variations in their execution time. Therefore, in this paper we propose a new non-iterative algorithm that results in a similar bandwidth distribution but with significant benefits in overhead and we compare it with the approach in [36]. The new algorithm takes advantage of the fact that, if any VM will reach its U_{lax} limit by using Equation 6, then the first to reach that limit will be the VM with the lowest value of $U_{lax}(V_i)/w(V_i)$ (see [36]). By setting the U_{add} values for all VMs, starting with the VM with lowest $U_{lax}(V_i)/w(V_i)$ and ending with the largest, it is guaranteed that the first m VMs (m may be 0) will reach their U_{lax} limit, while all subsequent VMs will not reach this limit. A single iteration through all VMs is sufficient to achieve the same bandwidth distribution as the previously proposed algorithms with several iterations.

Algorithm 1 presents the pseudocode of the proposed bandwidth distribution. Only VMs that are enabled (not depicted) and could benefit from additional bandwidth are considered (Line 7). At the beginning of the distribution among VMs of the same criticality level we compute the sum of the weights (w^{Σ}) and U_{lax} (U^{Σ}_{lax}) of the considered VMs (Lines 9-10). If the available U_{spare} exceeds U^{Σ}_{lax} all VMs can be satisfied immediately (Lines 12-15). Otherwise, the algorithm iterates over all VMs in the order provided in V and assigns utilization shares based on the normalized weights (Lines 17-18). U_{add} is bounded to U_{lax} (Line 19-20).

The basic bandwidth distribution formula (Equation 6) is $\mathcal{O}(n)$, but since the algorithm requires the VMs to be sorted, the computational complexity becomes $\mathcal{O}(n \cdot log n)$. Nevertheless, note that an initial sorting can be done at design time. If we take into account that this algorithm needs to be re-executed every time the bandwidth distribution parameters change, and that in most (if not all) cases only the parameters of a single VM changes between one execution of the algorithm and the next, then the re-ordering algorithm merely needs to correct the previous order, which may be done in a single iteration through all VMs, resulting in an algorithm complexity of $\mathcal{O}(n)$. Finally, note that the number of VMs assigned to the same core is determining, not the total number of VMs executed on the multicore processor. This implies a relatively low number of VMs per processor, e.g., $n \leq 6$.

Algorithm 1 Bandwidth Distribution

Require: V (sorted regarding increasing U_{lax}/w)
1: $U_{spare} \leftarrow$ COMPUTE_U_SPARE(V)
2: **for all** χ (descending order) **do**
3: **if** $U_{spare} = 0$ **then**
4: exit()
5: **end if**
6: $w^{\Sigma}[\chi] \leftarrow 0$, $U^{\Sigma}_{lax}[\chi] \leftarrow 0$
7: $V[\chi] \leftarrow \{V_i | V_i \in V \wedge \chi(V_i) = \chi \wedge U_{lax}(V_i) > 0\}$
8: **for all** $V_i \in V[\chi]$ **do**
9: $w^{\Sigma}[\chi] \leftarrow w^{\Sigma}[\chi] + w(V_i)$
10: $U^{\Sigma}_{lax}[\chi] \leftarrow U^{\Sigma}_{lax}[\chi] + U_{lax}(V_i)$
11: **end for**
12: **if** $U_{spare} \geq U^{\Sigma}_{lax}[\chi]$ **then**
13: **for all** $V_i \in V[\chi]$ **do** $U_{add}(V_i) \leftarrow U_{lax}(V_i)$
14: **end for**
15: $U_{spare} \leftarrow U_{spare} - U^{\Sigma}_{lax}[\chi]$
16: **else**
17: **for all** $V_i \in V[\chi]$ **do**
18: $U_{add}(V_i) \leftarrow U_{spare} \cdot w(V_i)/w^{\Sigma}[\chi]$
19: **if** $U_{add}(V_i) > U_{lax}(V_i)$ **then**
20: $U_{add}(V_i) = U_{lax}(V_i)$
21: $w^{\Sigma}[\chi] \leftarrow w^{\Sigma}[\chi] - w(V_i)$
22: $U_{spare} \leftarrow U_{spare} - U_{add}(V_i)$
23: **end if**
24: **end for**
25: exit()
26: **end if**
27: **end for**

D. Temporal Isolation and Minimum Bandwidth Guarantee

In Section I, key requirements for hypervisor-based consolidation of real-time systems and their certification were introduced. The presented computation bandwidth management approach fulfills this level of temporal isolation:

- The hypervisor's scheduling guarantees that all guest systems receive sufficient computation time to meet their timing requirements. Schedulability is enforced by the appropriate server design (the server's supply bound function is equal or greater than the guest's demand bound function for all t, see Section V-B).
- Overruns within a VM provoke under no circumstances that other VMs violate their timing requirements, since a VM is never executed if the associated server ran out of budget (see Section V-B).

The approach does not realize a completely uninfluenced execution of the guest systems, since this strong degree of temporal isolation is irreconcilable with a dynamic transfer of budget among VMs.

A specified minimum bandwidth allocation is guaranteed for all VMs. The online bandwidth allocation for a VM V_i is realized as an addition of a dynamic part ($U_{add}(V_i)$) to a static part ($U_{min}(V_i)$). Algorithm 1 computes under all circumstances for all VMs a $U_{add}(V_i)$ that satisfies $0 \leq U_{add}(V_i) \leq U_{lax}(V_i)$. These bounds are a direct implication of Equation 6. From an implementation perspective, $U_{min}(V_i)$ and $U_{add}(V_i)$ are realized as different parameters in the VM control block and only the latter is modified at runtime. Therefore, it is precluded that the allocated bandwidth falls below $U_{min}(V_i)$, implying that an execution time budget of at least $\Theta_i = U_{min}(V_i) \cdot \Pi_i$ is allocated every period Π_i. This minimum budget is not touched by the dynamic bandwidth management. The adaptive bandwidth distribution might just add budget.

The combination of the provided degree of temporal isolation and the guaranteed minimum bandwidth allocations ensures the correct execution of the guests in terms of their timing constraints, independent from the execution of other VMs on the same core. Therefore, these guaranteed minimum bandwidths and the described temporal isolation are the basis for both the schedulability analysis and a potential certification. An actual certification would imply other complementary aspects such as the proper determination of the minimum bandwidths needed for guaranteed timely execution of each VM in due consideration of the inter-core interferences through shared memory and bus interference, but these aspects are orthogonal aspects with respect to this work and beyond the scope of this paper. Very pessimistic minimum bandwidths can be expected if these interferences are considered, but our approach has the benefit of reclaiming unused capacity at runtime, thus making an efficient use of the processor even in such a situation.

VII. Evaluation

A. Evaluation Platform

The approach was implemented on an IBM PowerPC 405 multicore processor [44], a 32-bit RISC core for low-cost and low-power embedded systems clocked at 300 MHz. It features a scalar 5-stage pipeline with single-cycle execution of most instructions, separate instruction and data caches (16KB each) as well as a memory management unit with a software-managed translation lookaside buffer.

The execution times were determined with the IBM PowerPC Multicore Instruction Set Simulator [45], which allows for cycle-accurate evaluations with low effort on different hardware configurations. The simulator models all processor resources including caches. Many components of the simulated hardware can be configured, for example, the number of cores or cache sizes. If the program does not perform external memory accesses, the execution is identical to execution on real hardware. In order to ensure this, the caches are pre-loaded: the examined software routines are executed completely out of instruction cache and use the data cache for data storage, resulting in completion of loads and stores in one cycle. The instruction set simulator includes a trace tool for WCET analysis, which shows all the instructions executed and keeps track of the number of cycles used.

B. Prototype

We integrated the approach into the real-time multicore hypervisor *Proteus* [46] and the real-time operating system *ORCOS* [1] for 32-bit multicore PowerPC 405 architectures.

[1] Proteus and ORCOS are free software, released under the GNU General Public License. The source code can be downloaded from http://orcos.cs.uni-paderborn.de/orcos/browser/ProteusMC .

TABLE I: Memory Footprint (2 VMs)

Feature	Memory Footprint [bytes]		
	text	data	total
Base Hypervisor	8224	2980	11204
Paravirtualization	252	148	400
Bandwidth Redistribution	1996	292	2288
Total	10472	3420	13892

Proteus is a symmetric hypervisor: all cores have the same role and execute guest systems. When the guest traps or calls for a service, the hypervisor takes over control and its own code is executed on that core. Different guests on different cores can perform this context switch from guest to hypervisor at the same time. The hypervisor's scheduler can as well be executed on different cores at the same time. The hypervisor supports both kinds of virtualization and the concurrent hosting of paravirtualized and fully virtualized guests is possible without restriction. The dynamic bandwidth adaptation is however confined to paravirtualized guests. Fully virtualized guests receive a constant bandwidth allocation ($U_{lax} = 0$).

Depending on the requirements of the application, Proteus can be configured statically. The base configuration requires a total of about 11KB.[2] The required addition of paravirtualization support accounts for 400B. The dynamic bandwidth redistribution functionality consisting of scheduler and module for the communication between hypervisor and OS add 2.3KB. If all features required for dynamic bandwidth management are enabled, the memory requirement of the hypervisor hosting two VMs sums up to about 14KB (see Table I). For each additional VM the memory requirement increases by 56B.

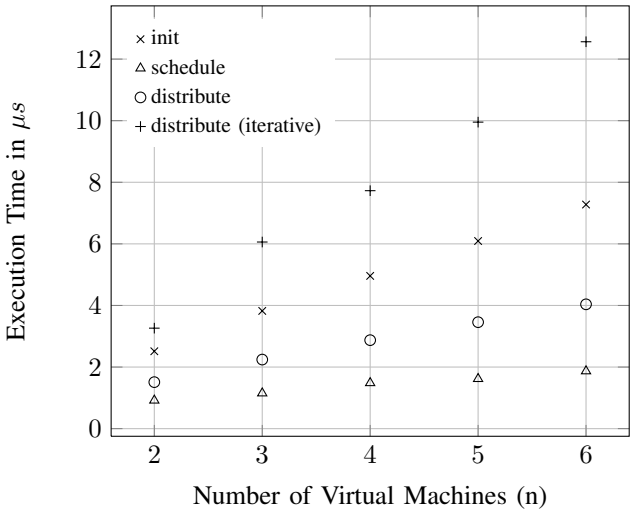
C. Paravirtualization Effort

In order to paravirtualize an OS for the presented adaptive approach, the scheduler has to be modified and a protocol-compliant communication with the hypervisor has to be added. The required communication between guest OS and hypervisor is realized by both hypercalls and shared memory communication. For the latter, a memory region within the memory space that is assigned to a VM is dedicated to paravirtualization communication. It is accessible by hypervisor and corresponding VM, however not by any other VM. The hypervisor informs the guest OS about bandwidth allocation changes via shared memory. Hypercalls are used by the guest OS to immediately invoke communication and pass control when it does not need the remaining assigned computation bandwidth in the current period and would otherwise idle or to inform the hypervisor about a change of U_{lax}.

The communication functionality is provided by a library. Main modification is the addition of the protocol-compliant passing of scheduling information to the hypervisor. Instead of idling, the guest OS should yield. In case of a task mode change, the guest OS has to inform the hypervisor. In order to detect whether the hypervisor changed the bandwidth allocation, the control flow has to be adapted: after a context switch from hypervisor to OS, the OS has to check a processor flag and if the allocation was modified read out the shared memory.

[2]All executables are generated with option -O2 for the GNU C compiler, which focuses on the performance and not primarily on the code size.

Fig. 2: Execution Times of Scheduler Routines Subject to the Number of Virtual Machines (PowerPC 405 @300 MHz)



In case of ORCOS, the paravirtualization effort accounted for 50 lines of C++ code.

D. Execution Times

Figure 2 depicts the execution times of the implemented routines from two to six VMs. The `init` function initializes the data structure for the server management and performs already an initial bandwidth distribution. It is called once at system start. `schedule` implements the scheduling policy: it determines (1) which VM to execute next and (2) for how long. `distribute` computes the additional bandwidth allocations based on the current resource requirements (Algorithm 1). The execution times are all in the range of about 1 to 8 microseconds. `init` requires the longest execution time. The most frequently called routine `schedule` is characterized by a low execution time below 2 microseconds. The execution time of `distribute` is for six VMs about 4 microseconds. For comparison purposes, we also show the execution times of an iterative version of the elastic bandwidth distribution referred as *distribute (iterative)*. A lower execution time for the new non-iterative algorithm can be observed; the larger n, the larger the difference (up to 58% for six VMs).

The execution time of all functions is dependent on the number of VMs n assigned to the same core. `init` and `distribute` have a computational complexity of $\mathcal{O}(n \cdot logn)$ (see Section VI-C), however, this is not observable for these small numbers of VMs, while it is $\mathcal{O}(n)$ for `schedule`.

The hypercall `vm_yield` (voluntarily release the processor) has an execution time of 507 ns, measured until the start of the hypervisor's `schedule` routine. By calling `sched_set_param`, the guest OS passes information to the hypervisor's scheduler, e.g. to inform about a mode change. The execution time of this hypercall is 793 ns, with the measurement stopped when the calling VM resumes its execution. The worst-case execution times for a shared memory read and write are 2.193 μs and 1.809 μs, respectively.

TABLE II: Thresholds for Distribution of Dynamic Slack (PowerPC 405 @300 MHz)

Slack Threshold $[\mu s]$ number of virtual machines				
n=2	n=3	n=4	n=5	n=6
2.250	4.493	5.123	5.707	6.287

E. Overhead vs. Benefit

The overhead of a bandwidth redistribution is the same for structural and dynamic slack. However, the frequency of the latter is much higher than for the former, since it is potentially invoked every time a VM yields and in the following VM capacity replenishment. Conversely, structural slack is expected to change possibly in a scale of seconds. Therefore, the bandwidth redistribution costs are crucial for taking advantage of the dynamic slack.

The total overhead consists of the overhead of a regular OS-to-hypervisor context switch plus the additional overhead of the hypercall `vm_yield`, the execution time of the function `distribute`, and the readout of the shared memory. Both the call to the hypervisor function `schedule` and the check of the processor flag by the guest OS have to be performed regardless of whether the hypervisor redistributes or not. With an execution time for a context switch from guest OS to hypervisor of 450 ns, the additional overhead of the hypercall `vm_yield` is only 57 ns. The costs for calling `distribute` and accessing shared memory dominate.

Table II lists the thresholds for the redistribution of dynamic slack as a function of the number of VMs: if the amount of dynamic slack is greater than the threshold, the benefits of a redistribution exceed the costs. In case of two VMs, the dynamic slack can be passed directly to the other VM, without having to call the distribute function, resulting in a significantly lower threshold. For the redistribution among three to six VMs, the amount of dynamic slack has to be greater than $4.5\mu s$ to $6.3\mu s$, respectively. We believe these are still low values to allow taking effective advantage of dynamic slack in many practical circumstances.

F. Comparison with other Approaches

There are three main approaches to provide temporal isolation of multiple VMs with real-time constraints:

1) dedicated processor for each real-time VM
2) static cyclic schedule with fixed execution time slices
3) execution-time servers

In the following, these three solutions are compared qualitatively. Subsequently, two kinds of the third solution are compared quantitatively, namely, servers with static bandwidth allocation and servers managed with the introduced adaptive approach.

In the first solution, a processor core hosts a single VM with real-time requirements. Other VMs without real-time requirements can execute in the background, not jeopardizing the response times of the real-time VM. This restricted approach does not use the full potential of virtualization and often leads to both a low utilization of the processors and a high number of required processors.

In the second approach, a static cyclic schedule [47] is designed by analyzing the guests' task sets and assigning execution time windows within a repetitive major cycle to the VMs based on the required utilization and execution frequency. This static scheduling approach is for example part of the software specification ARINC 653 [48] for avionics systems. It is well-analyzable but lacks run-time flexibility, thus being inadequate for cases with dynamic changes of the demands, since a reaction is only possible by redesigning the schedule. Such a redesign can seldom be computed online due to the high computation time overhead involved.

Finally, our work follows the third solution: the computation requirements of the VMs are abstracted as execution time servers, which are scheduled by the hypervisor as periodic tasks. The hypervisor's scheduler enforces the server bandwidths. Anyway, a periodic server with a fixed bandwidth [49] cannot react to mode changes and remains active when the associated guest system idles until its budget is exhausted.

In the following, the performance of two server-based approaches is compared, namely periodic servers with fixed and with adaptive bandwidth distribution as proposed in this work. The comparison is carried out through simulation with synthetically generated workloads. The real-time scheduling simulator RTSIM is used, developed at Retis Lab of the Scuola Superiore Sant'Anna [50]. However, RTSIM had to be extended for the purpose of our work, namely with server-based scheduling of virtual machines, the compared scheduling algorithms, criticality levels, and the possibility to generate synthetic workloads in addition to the existing manual input.

We used Brandenburg's toolkit SchedCAT [51], which is based on Emberson et al.'s [52] method for workload generation, to generate unbiased synthetic server sets with a given total utilization $\bar{U} = \sum_{i=1}^{n} U_{min}(V_i)$. As the proposed approach is a partitioned multicore scheduling solution, we analyze the scheduling of a set of servers assigned to one core. 1000 sets of VMs were generated according to the following parameter ranges:

- n uniformly distributed over $[2, 6]$
- $\bar{U}$ uniformly distributed over $[0.1, 0.2, ..., 0.7]$
- $U_{min}(V_i)$ uniformly distributed over $[0, \bar{U}]$
- Π_i generated as harmonic within $[10\mu s, 1000\mu s]$ (Θ_i follows as $\Theta_i = U_{min}(V_i) \cdot \Pi_i$)
- $U_{lax}(V_i)$ uniformly distributed over $[0, 0.20]$
- $p(V_i)$ uniformly distributed over $[0.05, 0.20]$
- $bdf(V_i)$ uniformly distributed over $[0.50, 1.00]$

A VM has randomly generated either only one or two modes. A $U_{lax}(V_i)$ is assigned to each mode. For simplicity, the weights are based on $U_{lax}(V_i)$. $p(V_i)$ denotes the probability of a mode change at the beginning of each period of V_i. The bandwidth demand factor $bdf(V_i)$ represents a variable demand of the server budget within one specific server period, assuming a value in the interval $[bdf(V_i) * \Theta_i, \Theta_i]$. If $bdf(V_i) = 1$, V_i needs the worst-case demand in this server period. A smaller value results in idle time, which might be

TABLE III: Virtual Machine Set I

VM	Period [us]	U_{min}	U_{lax} mode 1	U_{lax} mode 2	Criticality	p	bdf
VM1	1000	0.23	0.20	0.13	HI	0.2	0.87
VM2	100	0.05	0.12	0.04	HI	0.2	0.94
VM3	1000	0.14	0.03	0.11	HI	0.2	0.59
VM4	500	0.18	0.09	0.15	HI	0.2	0.72

TABLE IV: Virtual Machine Set II

VM	Period [us]	U_{min}	U_{lax} mode 1	U_{lax} mode 2	Criticality	p	bdf
VM1	100	0.17	0.20	0.0	HI	0.2	1.00
VM2	900	0.38	0.12	0.17	LO	0.05	1.00
VM3	300	0.25	0.08	0.12	LO	0.15	1.00

redistributed by the adaptive approach, but not by the fixed bandwidth management. Every configuration was simulated for ten hyperperiods, but limited to $10s$ of simulated time.

As it was shown in Section VI-D, the specified minimum bandwidths are guaranteed and this was confirmed in these multiple simulation runs. The bandwidth requirements of all virtual machines were fulfilled in all test scenarios.

The figures Fig. 3 and Fig. 4 show plots from actual execution traces based on the VM sets of Table III and Table IV in order to illustrate the adaptive distribution policy. In Fig. 3, the allocated bandwidths to four VMs over ten hyperperiods are plotted. In addition, dashed horizontal lines indicate the guaranteed minimum bandwidths. The probability of mode change was not generated randomly but set for alls VM to 0.2 in order to obtain a scenario with many bandwidth redistributions, resulting in 21 mode changes.

Fig. 4 allows for a closer look at the adaption process triggered by three mode changes (there is no dynamic slack in this experiment), both the point in time of enforcement and the influence of criticality and weight. At the beginning, VM1 and VM2 are in mode 1 and VM3 is in mode 2. VM1 receives the entire spare bandwidth of 0.2, since it is of higher criticality than VM2 and VM3 and it can use an additional bandwidth of 0.2 in its current mode. At t=700us (marked as (1) in the diagram), VM1 switches to mode 2, characterized by $U_{lax} = 0$, so that the spare bandwidth becomes available to VM2 and VM3. Both get a share of 50%, since their current modes have an equal U_{lax} and the weights are directly related to these values in this experiment. Both VMs idle already at this point in time, for which reason the additional bandwidth is not assigned until the next periods. At t=1400 (marked as (2) in the diagram), VM3's change from mode 2 to 1 results in a new balance between the weights and consequently in an increased bandwidth for VM2 at the expense of VM3. When VM1 changes back to mode 1 at t=2500 (marked as (3) in the diagram), the entire spare bandwidth is immediately assigned to this VM.

To assess the effectiveness of our mechanism of distributing slack bandwidth according to the presented policy, we define the *relative error* δ of the budget allocation, defined for the k^{th} period of the server that executes V_i based on the assigned

Fig. 3: Trace of the Bandwidth Assignment to 4 VMs over 10 Hyperperiods

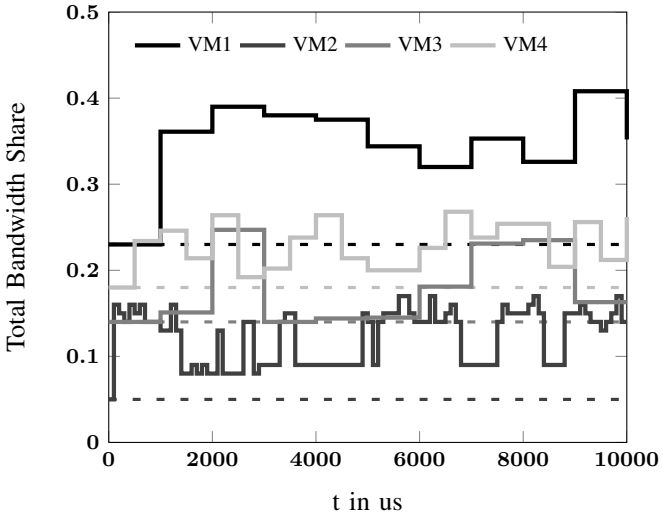
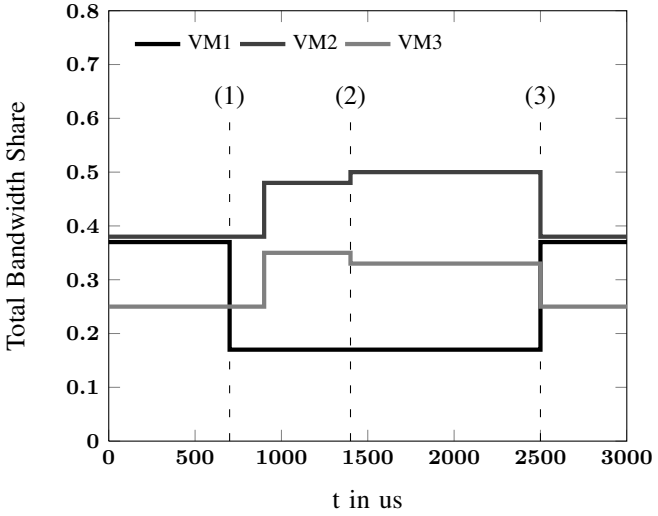


Fig. 4: Trace of the Bandwidth Assignment to 3 VMs: Detailed Look at Redistribution Triggered by Mode Changes



execution budget Θ_i^k and the actually desired budget Θ_i^{k*}:

$$\delta(V_i, k) = \frac{\Theta_i^k - \Theta_i^{k*}}{\Theta_i^{k*}} \tag{8}$$

The desired budget Θ_i^{k*} is defined by $U_{max}(V_i) = U_{min}(V_i) + U_{lax}(V_i)$ of the current mode and therefore not to be mistaken with the required budget $U_{min}(V_i)$ that guarantees schedulability. Including the bandwidth demand factor $bdf(V_i)$ as well, the desired budget is a random value within the following interval:

$$\Theta_i^{k*} \in [bdf(V_i) \cdot U_{max}(V_i) \cdot \Pi_i,\ U_{max}(V_i) \cdot \Pi_i] \tag{9}$$

In case of a negative $\delta(V_i, k)$, the desired budget was not saturated. A positive $\delta(V_i, k)$ denotes idle time of V_i in the considered period and therefore unused budget. Finally, $\delta(V_i)$

TABLE V: Relative Error δ of Allocated Bandwidth and Desired Bandwidth and Unused Bandwidth U_{unused} for Fixed Distribution and Adaptive Distribution

Policy	Average(δ)[%]	U_{unused}[%]
Fixed	9.6	25.2
Adaptive (Structural Slack)	4.1	16.7
Adaptive (Structural + Dynamic Slack)	2.1	12.4

is the average over all $|\delta(V_i, k)|$ for all periods k of VM $V_i \in V$. We keep track of the average values of each $\delta(V_i)$ and define δ as the average over all $\delta(V_i)$.

In a nutshell, the metric for this experiment is the relative error of allocated budget and desired budget. The desired budget changes constantly during runtime, based on both mode changes and a bandwidth demand that varies per period, that is to say that the VMs might not need the worst-case demand in a specific period. The smaller the relative error, the more effective the bandwidth allocation, since the relative error indicates either a non-saturated desired budget or an unused budget. The experiment with synthetically generated workloads investigates whether the adaptive approach is able to follow the varying computation bandwidth demands.

Table V lists the average values of δ and of the total unused utilization ($U_{unused} = U_{lub} - U_{allocated}$). In the fixed bandwidth policy, U_{spare} was distributed with a one-time static allocation based on the weights of the first mode. *Adaptive (Structural)* denotes the adaptive distribution of structural slack only, whereas *Adaptive (Structural + Dynamic)* includes both kinds of slack.

For any VM in all periods, the fixed bandwidth distribution results in an average δ difference of 9.6% between desired budget and allocated budget and a total unused computation bandwidth of 25.2%. When using the proposed adaptive bandwidth distribution, but redistributing structural slack only, the average δ and unused bandwidth fall significantly to 4.1% and 16.7%. As expected, these values fall even further when using in addition the adaptive bandwidth distribution of dynamic slack to 2.1% and 12.4%, respectively. These low values of δ confirm that the actual distribution of bandwidth follows closely the desired bandwidths, showing the effectiveness of our approach in enforcing an elastic distribution.

VIII. Conclusion

Hypervisor-based virtualization gained significant interest in the embedded domain in the last years, but with static resource management policies. An adaptive management of the computation bandwidth is an approach of great and so far untapped potential. This work proposed an adaptive bandwidth management for such systems, providing temporal isolation among virtual machines, defined not as an uninfluenced behavior, but as the guarantee that all guests are able to meet their timing constraints. Periodic execution time servers and the elastic task model combine analyzability at design time with adaptability at runtime. The correct execution of a virtual machine depends only on the server parameters and not on the behavior of other virtual machines, and is thus protected from potential overloads within another virtual machine.

This work explored dynamic budget replenishment as an efficient way to realize adaptive bandwidth reallocation in a multi-criticality setting, taking advantage of the slack generated by mode changes of virtual machines and shorter execution times than the declared worst-case. The bandwidth distribution is carried out with fine-grained control according to the elastic model, benefiting selected applications that can take advantage of higher computational bandwidth to produce improved results.

We claim that the use of the periodic resource model, Rate-Monotonic scheduling and a bandwidth distribution mechanism that ensures a guaranteed minimum bandwidth for all VMs is compatible with certification and thus the use of more efficient adaptive techniques in safety-critical settings. The proposed mechanisms were implemented and a quantification of the incurred overheads was carried out, showing their practicality and effectiveness in building more efficient embedded systems.

In future work, we plan to remove the constraint that only independent virtual machines are considered. If systems that have to communicate are consolidated, inter-VM communication is required. In addition, the influence of shared resource usage should be investigated.

Acknowledgment

This work was partially supported by the Portuguese Government through FCT grants CodeStream PTDC/EEI-TEL/3006/2012, Serv-CPS PTDC/EEA-AUT/122362/2010, EXPL/EEI-AUT/2538/2013, and SMARTS - FCOMP-01-0124-FEDER-020536 and by the German Federal Ministry for Education and Research within the project ARAMiS with the funding ID 01IS11035. The responsibility for the content remains with the authors. We thank Yuan Gao for supporting the work on the simulator.

References

[1] J.E. Smith and R. Nair, *Virtual Machines*. San Francisco, CA: Elsevier, 2005, pp. 369–443.

[2] A. Monot et al., "Multicore Scheduling in Automotive ECUs," in *Proc. Embedded Real Time Software and Systems (ERTSS)*, 2010.

[3] S. Baruah et al., "Mixed-Criticality Scheduling: Improved Resource-Augmentation Results," in *Proc. Conference on Computers and Their Applications (CATA)*, 2010, pp. 217–223.

[4] International Organization for Standardization, "ISO 26262 — Road vehicles — Functional safety, Part 1 - 10," Nov. 14, 2011.

[5] Z. Gu and Q. Zhao, "A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization," *Journal of Software Engineering and Applications*, vol. 5, no. 4, pp. 277–290, Jan. 2012.

[6] S. Groesbrink et al., "Fair Bandwidth Sharing among Virtual Machines in a Multi-criticality Scope," in *Proc. Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2013.

[7] L. Abeni et al., "QoS Management through Adaptive Reservations," *Real-Time Systems*, vol. 29, pp. 131–155, 2005.

[8] R. Santos et al., "On-line Schedulability Tests for Adaptive Reservations in Fixed Priority Scheduling," *Real-Time Systems*, vol. 48, pp. 601–634, June 2012.

[9] N. Khalilzad et al., "Bandwidth Adaption in Hierarchical Scheduling Uzing Fuzzy Controllers," in *Proc. Symposium on Industrial Embedded Systems (SIES)*, 2012, pp. 148–157.

[10] A. Block et al., "An Adaptive Framework for Multiprocessor Real-Time Systems," in *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2008, pp. 23–33.

[11] G. Buttazzo et al., "Resource Management on Multicore Systems: The ACTORS Approach," *IEEE Micro*, vol. 31, pp. 72–81, 2011.

[12] M. Maggio et al., "A Game-Theoretic Resource Manager for RT Applications," in *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2013, pp. 57–66.

[13] A. Zabos et al., "Spare Capacity Distribution Using Exact Response-time Analysis," in *Proc. International Conference on Real-time and Network Systems (RTNS)*, 2009, pp. 97–106.

[14] G. Lipari and S. Baruah, "Greedy Reclamation of Unused Bandwidth in Constant Bandwidth Servers," in *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2000, pp. 193–200.

[15] M. Caccamo et al., "Efficient Reclaiming in Reservation-based Real-time Systems With Variable Execution Times," *IEEE Transactions on Computers*, vol. 54, pp. 198–213, 2005.

[16] L. Nogueira and L. Pinho, "Capacity Sharing and Stealing in Dynamic Server-based Real-Time Systems," in *Proc. Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1–8.

[17] G. Bernat and A. Burns, "Multiple Servers and Capacity Sharing for Implementing Flexible Scheduling," *Real-Time Systems*, vol. 22, pp. 49–75, 2002.

[18] L. Marzario et al., "IRIS: A New Reclaiming Algorithm for Server-based Real-time Systems," in *Proc. Real-time and Embedded Technology and Applications Symposium (RTAS)*, 2004, pp. 211–218.

[19] K. Arzen et al., "Adaptive Resource Management Made Real," in *Proc. Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2011.

[20] H. Su and D. Zhu, "An Elastic Mixed-Criticality Task Model and Its Scheduling Algorithm," in *Proc. Design, Automation and Test in Europe (DATE)*, 2013, pp. 147–152.

[21] J. Anderson et al., "Multicore Operating-System Support for Mixed Criticality," in *Proc. Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[22] M. Mollison et al., "Mixed-Criticality Real-Time Scheduling for Multicore Systems," in *Proc. International Conference on Computer and Information Technology (CIT)*, 2010, pp. 1864–1871.

[23] J. Herman et al., "RTOS Support for Multicore Mixed-Criticality Systems," in *Proc. Real-Time Technology and Applications Symposium (RTAS)*, 2012, pp. 197–208.

[24] F. Bruns et al., "An Evaluation of Microkernel-based Virtualization for Embedded Real-time Systems," in *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2010, pp. 57–65.

[25] A. Crespo et al., "Partitioned Embedded Architecture based on Hypervisor: the XtratuM Approach," in *Proc. European Dependable Computing Conference (EDCC)*, 2010, pp. 67–72.

[26] E. Carrascosa et al., "XtratuM Hypervisor Redesign for LEON4 Multicore Processor," in *Proc. Workshop on Virtualization for Real-time Embedded Systems*, 2013.

[27] J. Yang et al., "Implementation of Compositional Scheduling Framework on Virtualization," *SIGBED Review*, vol. 8, pp. 30–37, March 2011.

[28] T. Cucinotta et al., "Providing Performance Guarantees to Virtual Machines Using Real-Time Scheduling," *Euro-Par Parallel Processing Workshops (Lecture Notes in Computer Science)*, vol. 6586, pp. 657 – 664, 2011.

[29] J. Lee et al., "Realizing Compositional Scheduling Through Virtualization," in *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 13–22.

[30] N. Navet et al., "Multi-source and Multicore Automotive ECUs - OS Protection Mechanisms and Scheduling," in *Proc. International Symposium on Industrial Electronics (ISIE)*, 2010, pp. 3734 – 3741.

[31] T. Liebetrau et al., "Energy Saving in Automotive E/E Architectures," Infineon Technologies, www.infineon.com, Tech. Rep., Dec. 2012.

[32] P. Barham et al., "Xen and the Art of Virtualization," in *Proc. Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 164–177.

[33] S. King et al., "Operating System Support for Virtual Machines," in *Proc. USENIX Annual Technical Conference*, 2003.

[34] J. Kiszka, "Towards Linux as a Real-Time Hypervisor," in *Proc. Real Time Linux Workshop*, 2011.

[35] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 44–61, 1973.

[36] R. Marau et al., "Efficient Elastic Resource Management for Dynamic Embedded Systems," in *Proc. Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2011, pp. 981–990.

[37] P. Emberson and I. Bate, "Extending A Task Allocation Algorithm for Graceful Degradation Of Real-Time Distributed Embedded Systems," in *Proc. Real-Time Systems Symposium (RTSS)*, 2008, pp. 270–279.

[38] I. Shin and I. Lee, "Compositional Real-Time Scheduling Framework with Periodic Model," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 30:1–30:39, 2008.

[39] R.I. Davis and A. Burns, "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems," *ACM Computing Surveys*, vol. 43, pp. 35:1–35:44, 2010.

[40] G. Lipari and E. Bini, "Resource Partitioning Among Real-Time Applications," in *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2003, pp. 151–158.

[41] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *Proc. Real-Time Systems Symposium (RTSS)*, 2003, pp. 2–13.

[42] L. Almeida and P. Pedreiras, "Scheduling within Temporal Partitions: Response-time Analysis and Server Design," in *Conference on Embedded Software (EMSOFT)*, 2004, pp. 95–103.

[43] G. Buttazzo et al., "Elastic Scheduling for Flexible Workload Management," *IEEE Transactions on Computers*, vol. 51, pp. 289–302, 2002.

[44] IBM, *PowerPC 405 Processor Core - Manual*, www.ibm.com, 2005 [Jan. 6, 2014].

[45] IBM Research, "IBM PowerPC 4XX Instruction Set Simulator," https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_4XX_Instruction_Set_Simulator_(ISS), Oct 2012 [Jan. 6, 2014].

[46] K. Gilles et al., "Proteus Hypervisor: Full Virtualization and Paravirtualization for Multi-Core Embedded Systems," in *Proc. International Embedded Systems Symposium (IESS)*, 2013, pp. 293–305.

[47] T. Baker and A. Shaw, "The Cyclic Executive Model and Ada," *Real-Time Systems*, no. 1, pp. 7–25, 1989.

[48] P. Prisaznuk, "ARINC 653 Role in Integrated Modular Avionics (IMA)," in *Proc. Digital Avionics Systems Conference (DASC)*, 2008, pp. 1.E.5:1–1.E.5:10.

[49] L. Sha et al., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," in *Proc. Real-Time Systems Symposium (RTSS)*, 1986, pp. 181–191.

[50] C. Bartolini and G. Lipari, "RTSIM," http://rtsim.sssup.it/, 2012 [Jan. 6, 2014].

[51] B. Brandenburg, "Schedcat: the Schedulability Test Collection and Toolkit," 2013 [Jan. 6, 2014]. [Online]. Available: https://github.com/brandenburg/schedcat

[52] P. Emberson et al., "Techniques for the Synthesis of Multiprocessor Tasksets," in *Proc. Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010.